The PyUtilib Component Architecture Reference Manual

William E. Hart[1] John Siirola[2]

February 1, 2010

[1]Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; wehart@sandia.gov

[2]Sandia National Laboratories, TODO, PO Box 5800, Albuquerque, NM 87185; jdsiiro@sandia.gov

Contents

Chapter 1

Introduction

1.1 Component Architectures

Component Based Software Engineering (CBSE) has become one of the leading approaches to developing software [3]. CBSE is an object oriented approach to software design, which is an accepted strategy for managing software complexity in large systems. Object oriented design is traditionally done using classes and class inheritance; classes define interfaces, which are extended and customized in subclasses using class inheritance.

The main idea behind CBSE is to separate the declaration of component interfaces from their implementation. This allows for a more flexible design of software components that further encourages modularity of component interface and definitions. Furthermore, this segregation allows for explicit management of the interaction between components. Thus, we can begin to imagine software components as commodities that can be integrated into applications in a much more flexible manner.

A variety of mature, general purpose environments have been developed for defining and managing compoments (e.g., the CORBA Component Model [2]). Although Python interfaces have been developed for some of these environments, a variety of native Python component environments have also been developed, including Zope [10], Envisage [1], Trac [8], yapsy [9] and SprinklesPy [6].

This report describes the PyUtilib Component Architecture (PCA). The PCA is an extension of the Trac component framework [8], and it is included in the PyUtilib software package [5]. The development of the PCA was motivated by the following requirements, which arose while working with components in several scientific computing applications:

- *Well-defined framework core*: Many component architectures are embedded in larger software frameworks (e.g. Zope), which makes it difficult to extract and use just the software packages related to the component architecture.

- *Non-Singleton components*: The computational science applications that motivate PCA require both singleton components (which have a single unique instance) and non-singleton components (which have many unique instances).

- *Namespaces*: Using components in large software projects requires management across multiple libraries. Namespaces are needed to effectively manage components in these complex software projects.

- *Caching*: Components need to support applications where component interfaces are called many times. Thus, caching of this interaction is needed to minimize the overhead of the component architecture.

- *Loading from EGGs*: Support for loading EGG files is invaluable in dynamic applications. Further, loading components from EGG files provides another level of modularity to the management of software applications.

Another motivation for the development of the PCA was the desire to employ a light-weight framework that could be easily deployed. Thus, dependencies on a general purpose environment was undesirable. Additionally, support for interface conversion was not deemed sufficiently important to justify the use of a more complex framework like Zope or Envisage that supports this functionality.

1.2 Outline

The remainder of this report is divided into three sections. Chapter 2 provides a detailed description of the PCA. This includes both a tutorial introduction to the use of PCA classes, but also a description of the design objectives that guided the design of PCA. Chapter 3 describes PyUtilib extension packages that support specific components based on the PCA. This chapter motivates these packages and provides examples of their use. Finally, Chapter 4 describes how to deploy a PCA package that supports components that are dynamically loaded.

Chapter 2

The PyUtilib Component Architecture

The PyUtilib Component Architecture (PCA) is an extension of the Trac plugin framework [8] that is included in the PyUtilib software package [5]. The PCA was initially motivated by the need to have a Trac-like plugin framework that was self-contained. The core of PCA is provided by PyUtilib's `pyutilib.component.core` package, which contains a small set of Python classes. This chapter provides a tutorial introduction of the PCA as well as a detailed description of the PCA classes and their functionality.

2.1 PCA Definitions

There are different notions of software components [4], so we begin by providing some definitions. A *plugin* is a class that implements a set of related methods in the context of an application. Thus, a plugin can be described as a component definition. A *service* is an instance of a plugin class, so services are component instances. A service can be an instance of either a singleton or non-singleton plugin. There is at most one service for a singleton plugin, whereas there can be multiple services of non-singleton plugins.

An *interface* class defines a type that a plugin uses to register is capabilities. A plugin class includes declarations that denote that it implements one-or-more interfaces. An interface is defined by the methods and data that are used. However, the PCA does not enforce this interface or support interface conversions (see Zope [10] and Envisage [1] for examples of plugin frameworks that support this functionality).

A software application can declare *extension points* that other components can *plug in* to. An extension point is defined with respect to a specific *interface* class. Thus, a service that supports an interface plugs into an extension point for that interface. In this way, extension points provide a generic mechanism for components to employ the functionality provided by other services.

This mechanisms supports a flexible, modular programming paradigm that enables software applications to be extended in a dynamic manner. Extension points and the extensions contributed to them are stored in a global registry, and execution of these extensions is handled in a consistent manner. Thus, an application developer can define extension points without knowing how they will be implemented, and extension developers can register extensions without needing to know the details of how they are employed.

The general design of PCA is adapted from Trac [8]. The PCA generalizes the Trac component architecture by supporting namespace management of plugins, as well as non-singleton plugins. For those familiar with Trac, the following classes roughly correspond with each other:

Trac Class Name	PyUtilib Class Name
Interface	Interface
ExtensionPoint	ExtensionPoint
Component	SingletonPlugin
ComponentManager	PluginEnvironment

The `PluginEnvironment` class is used to manage plugins, but unlike Trac this class does not need to be explicitly constructed.

Additionally, PCA includes a global component registry, as well as a framework for automating the execution of components that match a given interface. This capability facilitates the dynamic registration and application of components within large software systems.

2.2 Core Plugin Classes

The PCA consists of the following core classes:

pyutilib.component.core.Interface Subclasses of this class declare plugin interfaces that are registered in the PCA.

pyutilib.component.core.ExtensionPoint A class used to declare extension points, which can access services with a particular interface.

pyutilib.component.core.Plugin Subclasses of this class declare plugins, which can be used to implement interfaces within the PCA.

pyutilib.component.core.SingletonPlugin Subclasses of this class declare singleton plugins, for which a single service is constructed.

pyutilib.component.core.PluginEnvironment A class that maintains the registries for interfaces, extension points, plugins and services.

pyutilib.component.core.PluginGlobals A class that maintains global data concerning the set of environments that are currently being used.

pyutilib.component.core.PluginError The exception class that is raised when errors arise in the PCA.

2.2.1 Interfaces and Extension Points

A subclass of the `Interface` class is used to declare extension points in an application. The `ExtensionPoint` class is used to declare an extension point and to retrieve information about the plugins that implement the specified interface. For example, the following is a minimal declaration of an interface and extension point:

```
class IMyInterface(Interface):
    """An interface subclass"""

ep = ExtensionPoint(IMyInterface)
```

Note that the `IMyInterface` class is not required to define the API of the interface. The PCA does not enforce checking of API conformance for plugins, and hence any declaration in the `IMyInterface` class would be ignored. Additionally, note that an instance of `IMyInterface` is not created; the `IMyInterface` class is simply used to declare a type that is used to index related plugins.

An instance of `ExtensionPoint` can be used to iterate through all extensions, or to search for an extension that matches a particular keyword. For example, the following code iterates through all extensions and applies the `pprint` method:

```
for service in ep:
  service.pprint()
```

If you wish to know the number of services that are registered with an extension point, you can call the standard `len` function:

```
print len(ep)
```

Several other methods can be used to more carefully select services from an extension point. The `extensions` method returns a Python `set` object that contains the services:

```
#
# This loop iterates over all services, just the same as when an
# the iterator method is used (see above).
#
for service in ep.extensions():
  service.pprint()
```

The Python `__call__` method provides a convenient shorthand for this same function. Thus, the following is equivalent:

```
for service in ep():
  service.pprint()
```

These methods have two optional arguments that control the selection of services. The `all` keyword indicates whether the set returned by `extensions` includes all disabled services.

```
#
# This loop iterates over all services, including the 'disabled'
# services.
#
for service in ep.extensions(all=True):
  service.pprint()
```

It is convenient to explicitly support enabling and disabling services in many applications, though services are enabled by default. Disabled services remain in the registry, but by default they are not included in the set returned by an extension point.

The PCA can also support *named services*, which requires that the services have a `name` attribute. Service names are not required to be unique. For example, when multiple instances of a non-singleton plugin are created, then these services can be accessed as follows:

```
#
# A simple plugin that implements the IMyInterface interface
#
class MyPlugin(Plugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="myname"

#
# Another simple plugin that implements the IMyInterface interface
#
class MyOtherPlugin(Plugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="myothername"

#
# Constructing services
#
service1 = MyPlugin()
service2 = MyPlugin()
service3 = MyOtherPlugin()

#
```

```
# A function that iterates over all IMyInterface services, and
# returns the MyPlugin instances (which are service1 and service2).
#
def get_services():
    ep = ExtensionPoint(IMyInterface)
    return ep("myname")
```

In some applications, there is a one-to-one correspondence between service names and their instances. In this context, a simpler syntax is to use the `service` method:

```
class MySingletonPlugin(SingletonPlugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="mysingletonname"

ep = ExtensionPoint(IMyInterface)
ep.service("mysingletonname").pprint()
```

The `service` method raises a `PluginError` if there is more than one service with a given name. Note, however, that this method returns `None` if no service has been registered with the specified name.

Note that an integer cannot be used to select a service from an extension point. Services are not registered in an indexable array, so this option is disallowed.

2.2.2 Plugins

PCA plugins are subclasses of either the `Plugin` or `SingletonPlugin` classes. Subclasses of `Plugin` need to be explicitly constructed, but otherwise they do not need to be registered; simply constructing a subclass of `Plugin` invokes the registration of that instance. Similarly, simply declaring a subclass of `SingletonPlugin` invokes both the construction and registration of this component.

PCA plugins are registered with different interfaces using the `implements` function, which is a static method of `Plugin`. Note that a plugin can be registered with more than one interface. Further, a service can be applied to different extension points independently, but it can maintain state information that impacts its use across different extension points.

Although the default behavior of the PCA is to ignore the declarations in an interface class, the `implements` function includes an `inherit` keyword can be used to define a plugin that inherits interface methods. For example:

```
class IMyInterface(Interface):
    def print(self):
        print "This is the default print method"
```

```
    def add(self, x):
        return x+2

class MyPlugin(Plugin):
    implements(IMyInterface, inherit=True)

    def add(self, x):
        return x+3
```

In this example, we the `MyPlugin` class implements the `IMyInterface` interface. Since the `inherit` keyword is `True`, the `MyPlugin` class inherits the `print` method. Thus, `MyPlugin` has a complete implementation of the `IMyInterface` interface.

Although this behavior is generally useful, the API for PCA intentionally does not make interface inheritance the default behavior. When inheritance is used, a developer can get into trouble if they mistype the name of a plugin method. When this occurs, the interface method is used, without any notification to the user. This could easily lead to erroneous plugin behavior that is quite difficult to track down.

2.2.3 Environments

The `PluginEnvironment` class defines a namespace that contains component services and interfaces. These namespaces provide a mechanism for organizing component services in an extensible manner. Applications can define new namespaces that contain their services without worrying about conflicts with services defined in other Python libraries.

A global registry of environments is maintained by the `PluginGlobals` class. This registry is a stack of environments, and the top of this stack defines the current environment. When an interface is declared, its namespace is the name of the current environment. For example:

```
#
# Declare an interface in the current environment
#
class Interface1(Interface):
   pass

#
# Set the current environment to 'new_environ'
#
PluginGlobals.push_env( "new_environ" )

#
# Declare an interface in the 'new_environ' environment
```

```
#

class Interface2(Interface):
    pass

#
# Go back to the original environment
#
PluginGlobals.pop_env()
```

Component services are automatically registered in namespaces in two ways. First, for each interface that the service implements, the service is registered in the namespace in which the interface was declared. Second, a service is registered in the namespace in which its corresponding plugin class is declared.

For example, consider the code in Figure 2.1. When `Plugin1` is instantiated, this service is registered in the following environments:

```
    env1 for Interface1
    env2 for Interface2
    env4 for Interface1
    env3
```

The last registration is the default, since a service is always registered in the environment where its plugin class is declared. Note that `env4` namespace is declared explicitly in this example.

2.2.4 Global Component Data

Global component data in PCA is managed in the `PluginGlobals` class. This class contains a variety of static methods that are used to access this data:

default_env This method returns the default environment, which is constructed when the PCA is loaded.

env This method returns the current environment if no argument is specified. Otherwise, it returns the specified environment.

push_env,pop_env These methods respectively push a new environment onto the environment stack and pop the current environment from the stack.

services This method returns the component services in the current environment (or the named environment if one is specified).

singleton_services This method returns the singleton component services in the current environment (or the named environment if one is specified).

```
#
# Declare Interface1 in namespace env1
#
PluginGlobals.push_env("env1")

class Interface1(Interface):
    pass

#
# Declare Interface2 in namespace env2
#
PluginGlobals.push_env("env2")

class Interface2(Interface):
    pass

PluginGlobals.pop_env()

#
# Declare Plugin1 in namespace env3
#
PluginGlobals.push_env("env3")

class Plugin1(Plugin):

    implements(Interface1)
    implements(Interface2)
    implements(Interface1,"env4")

PluginGlobals.pop_env()
```

Figure 2.1: Illustration of how plugin declarations are related to component environments.

load_services Load services using `IPluginLoader` extension points (see Section 3.1).

pprint This method provides a text summary of the registered interfaces, plugins and services.

clear This method empties the environment stack and defines a new default environment. This setup then bootstraps the configuration of the `pyutilib.component.core` environment. Note that this does not clear the component registry; in practice that may not make sense since it is not easy to reload modules in Python.

2.3 A Simple Example

Figure 2.2 provides a simple example that is adapted from the description of the Trac component architecture [7]. This example illustrates the three main steps to setting up a plugin:

1. Defining an interface
2. Declaring extension points
3. Defining classes that implement the interface.

In this example, a singleton plugin is declared, which automatically registers the component service. Non-singleton plugin services need to explicitly created, but they are also automatically registered.

If the script in Figure 2.2 is in the `task.py` file, then the following Python script illustrates how this plugin is used:

```
from task import *

# Construct a TaskList object and then add several tasks.
task_list = TaskList()
task_list.add('Make coffee', 'Really need to make some coffee')
task_list.add('Bug triage',
        'Double-check that all known issues were addressed')
```

This script generates the following output:

```
Task: Make coffee
      Really need to make some coffee
Task: Bug triage
      Double-check that all known issues were addressed
```

```python
# A simple example that manages a task list.  An observer
# interface is used to add actions that occur when a task
# is added.
from pyutilib.component.core import *

# An interface class that defines the API for plugins that
# observe when a task is added.
class ITaskObserver(Interface):

        def task_added(name, description):
            """Called when a task is added."""

# The task list application, which declares an extension point
# for observers.  Observers are notified when a new task
# is added to the task list.
class TaskList(object):
        observers = ExtensionPoint(ITaskObserver)

        def __init__(self):
            """
            The TaskList constructor, which initializes the list
            """
            self.tasks = {}

        def add(self, name, description):
            """Add a task, and notify the observers"""
            self.tasks[name] = description
            for observer in self.observers:
                observer.task_added(name, description)

# A plugin that prints information about tasks when they
# are added.
class TaskPrinter(SingletonPlugin):
        implements(ITaskObserver)

        def task_added(self, name, description):
            print 'Task:', name
            print '      ', description
```

Figure 2.2: A simple example of the Python Component Architecture

Chapter 3

PCA Extensions

In addition to the core component framework, PCA includes implementations for a variety of components that support commonly used functionality. These extensions of PCA are available in PyUtilib packages separate from the PCA core. This emphasizes the modularity of the PCA, and it illustrates how to define PCA components that are automatically registered as part of an application. The following sections briefly describe these PCA extensions.

3.1 Component Loaders

PCA components can be loaded from either Python modules or Python eggs. This capability supports the runtime extension of the PCA, which has proven very powerful in frameworks like Trac. Component services for loading are defined in the `pyutilib.component.loader` package. The core PCA defines extension points that use these services, which can be applied as follows:

```
import sys
import os
env = sys.environ["PATH"]
PluginGlobals.load_services(path=env.split(os.sep))
```

In this example, the user's `PATH` environment is used to define the list of directories that are searched for Python modules and eggs.

The `load_services` takes two other optional arguments that control how components are loaded. The `name_re` argument can be used to define a regular expression that filters the files in the directories that are searched. The following shows how to specify that services starting with *my* are loaded:

```
PluginGlobals.load_services(name_re="my.*")
```

By default, when services are loaded they are disabled. This facilitates the management of services in complex applications using configuration files (see below). The `auto_disable` flag can be used to automatically activate services:

```
PluginGlobals.load_services(auto_disable=False)
```

3.2 Registering Executables

The `pyutilib.component.executable` package defines the `ExternalExecutable` plugin, which is used to define services that provide information about external executables. Services declare the executable name and user documentation, and then service methods indicate whether the executable is enabled (i.e. whether it is found, and the path of the executable:

```
service = ExternalExecutable(name='ls',
            doc='A utility to list file in Unix operating systems')

service.enabled()
# Returns True if the executable is found on the user path.

service.get_path()
# Returns a string that defines the path to this executable,
# or None if service is disabled.
```

The registration process is simplified with the `pyutilib.services` module, which includes the `register_executable` function:

```
import pyutilib.services

pyutilib.services.register_executable('zip')
```

This function searchers the user's `PATH` environment for the `zip` executable (or `zip.exe` on Windows machines).

A developer can use the `registered_executable` function to the access the absolute path of a registered executable. If the executable is not found in the user's `PATH`, then this returns `None`. Also, if no executable is specified, then this function returns a list of all registered executables.

3.3 Temporary Files

The `pyutilib.component.config` packages provides a services for managing temporary files. The `TempfileManager` object is a component service whose methods can be used to create and cleanup temporary files; for convenience, this object is accessible from the `pyutilib.services` module.

The main method in this service is `create_tempfile`, which can create a temporary file with a specified suffix and prefix in a specified directory:

```
import pyutilib.services

pyutilib.services.create_tempfile(prefix='myfile',
                                  suffix='.txt', dir='/home/jdoe')
```

By default, this service creates unique filenames. However, if the `sequential_files` method is called, then the body of the temporary files will be an integer that is incremented every time a temporary file is created. Although these filenames may not be unique, this sequential naming scheme may make it easier to diagnose errors in a complex application.

This service keeps track of the temporary files that it creates. This allows an application developer to avoid this bookkeeping, and instead rely on this service to delete temporary files with the `clear_tempfiles` method. Furthermore, a developer can explicitly declare a file as temporary using the `add_tempfile` method, thereby allowing this service to delete it.

3.4 Options and Configuration Files

The `pyutilib.component.config` package defines interfaces and plugins for managing service options. The `Configuration` service is used to manage the global configuration of all services. This class coordinates with `Option` services. Plugins can declare options with the `declare_option` method, which registers these options with the `Configuration` service. This service reads and writes options to configuration files (using Python's `ConfigParser` package).

This package also declares the `ManagedPlugin` and `ManagedSingletonPlugin` classes, which are plugin base classes that include options that can be used to enable or disable services using the `Configuration` service.

3.4.1 Configuration Files

A PCA configuration file consists of a list of sections. Each section is lead by a `[section]` header, and a section contains a list of `name = value` entries. For example, the following configuration file consists of two sections with four option values:

```
# COMMENT
[globals]
a = 1
b = /dev/null
c = 1,2,3
[a.b]
zz = 4.5
```

PCA plugin classes declare component options with the `declare_option` method, and these options can be initialized with a configuration file. For example, the following plugin declares four options.

```
class PluginWithOptions(Plugin):
    def __init__(self):
        declare_option("a")
        declare_option("b")
        declare_option("c")
        declare_option("zz",section='a.b')
```

The default configuration section for an option is `[globals]`, but the option declaration can specification the section name. For example, the configuration file described above can be used to initialize the `PluginWithOptions` plugin.

The `Configuration` class manages loading and storing configuration data for the options that are registered by PCA services. This class defines the following methods:

clear Clear the configuration data.

load Load configuration data from a file.

pprint Print a simple summary of the configuration data.

save Write configuration data to a file.

sections Return a list of the sections that have been loaded.

summarize Summarize the options that have been registered with the PCA.

Once data is loaded with the `load` method, the `sections` method can be used to provide a list of the sections that were loaded. The Python `__contains__` method can also be used to check if a section was loaded:

```
from pyutilib.component.config import *

config = Configuration()

# Load configuration data
config.load('config.ini')

# Check if the 'globals' section was loaded
if 'globals' in config:
    print "The 'globals' section was loaded"

# Get the 'globals' section
section = config['globals']
```

The Python `__getitem__` method is used at the end of this example to get the data for the 'globals' section. Section data consists of a dictionary that maps option names to value strings. Note that the `Configuration` class automatically loads this data into the corresponding options that have been registered with the PCA.

3.4.2 Declaring Options

The `declare_option` creates an `Option` object that is a data member in a plugin. The standard syntax for this function is to specify the option name, which is used to define an attribute in the plugin with the same name:

```
class TmpPlugin(SingletonPlugin):

  declare_option('x')
```

The `local_name` keyword can be used to specify a different name for this attribute within the Plugin. For example, consider:

```
class TmpPlugin(SingletonPlugin):

  declare_option('x', local_name='y')
```

In this example, the option is declared with name `x` in the PCA registry, but it has attribute name `y` within this plugin. The `default` keyword defines the default value of an option, and the `doc` option specifies a document string that describes the option; this information is used by the `Configuration` class when printing option summary information.

As noted earlier, the `section` keyword can be used to specify the section in configuration data that this option is expected. The `section_re` keyword supports a more generic mechanism. If the `section_re` is specified with a regular expression, then this option will be initialized from any section that matches this regular expression. If sections match and contain data for this option, then the last section specified in the configuration data will be used to initialize this option.

Finally, the `cls` keyword specifies the option type. Option types are described in the next section.

3.4.3 Options Types

The default option type is `Option`. These options treat option values as strings, even when they could be interpreted as numeric values. The `BoolOption`, `IntOption` and `FloatOption` types respectively interpret option values as booleans, integers and floating point values. The `OptionError` exception is raised if the option value is not the appropriate value.

The `FileOption` type interprets the option value into a path. A relative path is converted to an absolute path using the path for the configuration file. Thus, a user can load file path data from any directory but specify the file data relative to the path of the application configuration. However, this option type also supports a `directory` keyword that can be used to specify how relative paths are resolved. The `ExecutableOption` type is an extension of `FileOption` that confirms that the file can be executed. If the file name does not include path information, then PCA will search for the executable using the user's `PATH` environment before initializing this option.

The `DictOption` type supports an interface to all options in a section. For example, consider:

```
class TmpPlugin(Plugin):

    options = DictOption(section="bar")
```

The `options` object will be populated by all data in the `bar` section. For example, if the names `a` and `b` are defined in this seciton, then they can be referenced as `options.a` and `options.b`. Similarly, data can be inserted into the `bar` section by simply specifying the value of attributes of this object:

```
options.c = 2
```

3.4.4 Using Options in Services

It is convenient for singleton plugins to declare options as part of the class definition:

```
class Plugin1(SingletonPlugin):

  declare_option("x")
```

This type of declaration makes sense since there is a single instance of the class `Plugin1`. For non-singleton plugins, this type of declaration would make the *same* option data available to all instances of the plugin. To declare different options for different non-singleton plugin instances, it suffices to execute `declare_option` within the plugin constructor:

```
class Plugin2(Plugin):

    def __init__(self, sec):
        declare_option("x", section=sec)
```

Note that this example allows the `Plugin2` services to distinguish the configuration of these different `x` options by specifying different sections in the configuration file. Although this is not required, this is often desirable in practice.

3.4.5 Managed Services

The PCA supports explicit management of services using the configuration management technology described in this section. The `ManagedPlugin` and `ManagedSingletonPlugin` classes include an `enable` option that controls whether their corresponding services are activated. When services are loaded from EGG files and modules, they are disabled by default. The `Services` section of a configuration file can be used to activate these plugins:

```
[Services]
plugin1 = True
plugin3 = True
```

This allows an application administrator to install a variety number of application services that a user selectively enables.

3.5 Other Extensions

The following extension packages include plugins and applications interfaces for PCA applications. These extension packages are less mature, and consequently they are not documented in detail right now.

pyutilib.component.loader This extension package defines a plugin that manages logging of PCA actions.

pyutilib.component.app This package defines a simple application class that can be used as the basis of a component-based application. This application class provides support for managing configuration from a configuration file, and for managing logging activity.

Chapter 4

Deploying PCA Packages

4.1 Background

The Python `setuptools` package is the *de facto* standard for deploying Python software. This package extends Python's `distutils` functionality. A key element of this extension is the `easy_install` command, which allows the installation of Python software from remote repositories. In particular, the Python Package Index (PyPI) provides a convenient repository for hosting Python packages. The `easy_install` command can easily upload and download packages from PyPI, thereby simplifying the distribution of Packages like Coopr, which depends on a variety of freely available packages.

The PCAleverages features of the `setuptools` package to facilitate the integration of Python packages that contain components. Here are the details:

- The `pyutilib.component` subpackage is a *namespace package*. Namespace packages are a mechanism for splitting a single Python package across multiple directories on disk. This allows different Python packages to provide components in separate subpackages (e.g. `pyutilib.component.mine` and `pyutilib.component.yours`).

- Subpackages in PyUtilib have been setup to dynamically load components that are registered in `pyutilib.component` packages. This leverages the `pkg_resources` package that included with `setuptools` by defining *entry points* for the Python packages that are loaded under the `pyutilib.component` namespace.

Although configuring PyUtilib to leverage these capabilities requires some black magic, we hope that PyUtilib developers will not need to delve into the details of this mechanism. The following section provides some guidelines for configuring a package such that its components are automatically loaded when PyUtilib is imported.

4.2 Example: pyutilib.component.loader

The `pyutilib.component.loader` package illustrates the basic organization that is needed to seamlessly integrate PCA components that are defined in an external Python package.

There are a variety of important details, which we enumerate in the following sections.

4.2.1 Directory Structure

The trunk version of the `pyutilib.component.loader` package has the following directory structure:

```
setup.py
pyutilib/
pyutilib/__init__.py
pyutilib/component/
pyutilib/component/__init__.py
pyutilib/component/loader/
pyutilib/component/loader/__init__.py
pyutilib/component/loader/plugin_eggLoader.py
pyutilib/component/loader/plugin_importLoader.py
```

A key aspect of this directory structure is that it mimics the structure in PyUtilib. Further, the files `pyutilib/__init__.py` and `pyutilib/component/__init__.py` must have the following definitions to ensure that `pyutilib.component` is a namespace package:

```python
# this is a namespace package
try:
    import pkg_resources
    pkg_resources.declare_namespace(__name__)
except ImportError:
    import pkgutil
    __path__ = pkgutil.extend_path(__path__, __name__)
```

4.2.2 PCA Modules

There are few restrictions on the content of the modules in `pyutilib/component/loader`. However, automatic loading of components requires that their associated Python modules are imported by `pyutilib/component/loader/__init__.py`. In this package, components are defined in the `plugin_eggLoader.py` and `plugin_importLoader.py` modules, which are imported by the `__init__.py` module.

4.2.3 Package Configuration

The `setuptools` package uses the `setup.py` module to configure the installation of `pyutilib.component.loader`. Figure 4.1 contains the listing of this file. Only a handful of these arguments are specific to an installation with `setuptools`:

- The `packages` options lists all package directories that are included in this package.
- The `namespace_packages` options lists all package directories that are namespace packages. There are two namespace packages in PyUtilib that need to be specified: `pyutilib` and `pyutilib.component`.
- The `entry_points` option specifies how components of this package are registered with `setuptools`. The `entry_points` option specifies a dictionary. The keys of this dictionary are *group names* that specify a set of related entry points. PyUtilib uses this dictionary to load packages that have been installed with `setuptools`. By convention, the PCA packages load plugins with the `pyutilib.component` group name.

As this example illustrates, an entry point relates an *entry name* with a package in the software repository. Although these entry names must be unique, PyUtilib does not rely on them having any particular syntax or semantics. Instead, PyUtilib simply loads each entry point within a given group. This triggers the registration of component plugins and services within PyUtilib, which is the desired result.

4.2.4 Uploading Package Releases

Once your Python package is ready for a release to other users, you can upload it to the PyPI repository using the following command:

```
python setup.py sdist upload -s
```

Note that before you upload the first time you will need to register your package with the following command:

```
python setup.py register
```

```python
import os
from setuptools import setup


def read(*rnames):
    return open(os.path.join(os.path.dirname(__file__),*rnames)).read()


setup(name="pyutilib.component.loader",
    version='3.2',
    maintainer='William E. Hart',
    maintainer_email='wehart@sandia.gov',
    url = 'https://software.sandia.gov/svn/public/pyutilib/'+
                                      'pyutilib.component.loader',
    license = 'BSD',
    platforms = ["any"],
    description = 'PyUtilib components for loading external packages',
    long_description = read('README.txt'),
    classifiers = [
        'Development Status :: 4 - Beta',
        'Intended Audience :: End Users/Desktop',
        'License :: OSI Approved :: BSD License',
        'Natural Language :: English',
        'Operating System :: Microsoft :: Windows',
        'Operating System :: Unix',
        'Programming Language :: Python',
        'Programming Language :: Unix Shell',
        'Topic :: Scientific/Engineering :: Mathematics',
        'Topic :: Software Development :: Libraries :: Python Modules'],
      packages=['pyutilib', 'pyutilib.component',
                             'pyutilib.component.loader'],
      keywords=['utility'],
      namespace_packages=['pyutilib', 'pyutilib.component'],
      entry_points = {
        'pyutilib.component' : [
            'component.loader = pyutilib.component.loader'
        ]
      },
      install_requires=['pyutilib.component.core',
                         'pyutilib.component.config']
      )
```

Figure 4.1: The `setup.py` file used by `pyutilib.component.loader`.

Acknowledgements

We thank Jean-Paul Watson for feedback on the design of the PyUtilib Component Architecture. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Bibliography

[1] *EnvisageCore*. https://svn.enthought.com/enthought/wiki/EnvisageThree/core.html, 2009.

[2] O. M. Group, *CORBA component model specification, version 4.0*, tech. rep., Object Management Group, Inc., 2006.

[3] G. Heineman and W. Councill, eds., *Component-Based Software Engineering, Putting the Pieces Together*, Addison-Wesley, 2001.

[4] R. Marvie, *Picolo: A simple python framework for introducing component principles*, in Euro Python Conference 2005, June 2005.

[5] *PyUtilib: A python utility library*. http://software.sandia.gov/pyutilib, 2009.

[6] *SprinklesPy*. http://termie.pbworks.com/SprinklesPy, 2009.

[7] *Trac component architecture*. http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture, 2009.

[8] *Trac*. http://trac.edgewall.org/, 2009.

[9] *yapsy*. http://yapsy.sourceforge.net/, 2009.

[10] *Zope*. http://www.zope.org/, 2009.